

springCloud+springCloudAlibaba



SpringCloud== 分布式微服务的一站式解决方案，多种微服务架构落地技术集合体，俗称微服务全家桶。



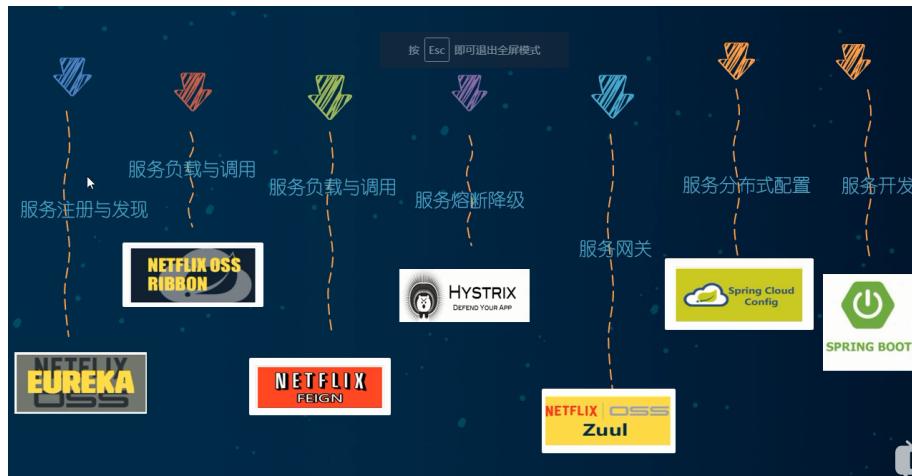
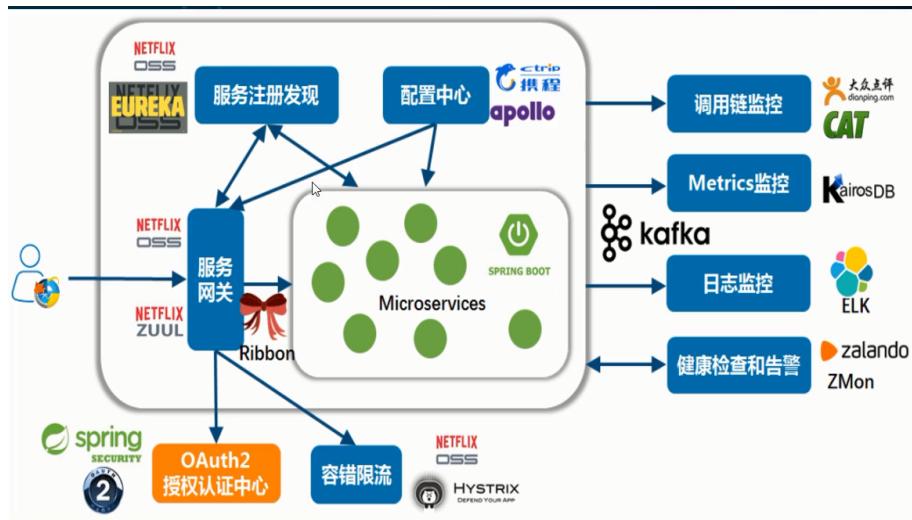
京东微服务：



阿里：



主流微服务架构：



版本选择

springboot 2.x版 springcloud H版

cloud Hoxton.SR1

boot 2.2.2.RELEASE

cloudalibaba 2.2.1.RELEASE

java 8

Maven 3.5及以上

Mysql 5.7及以上

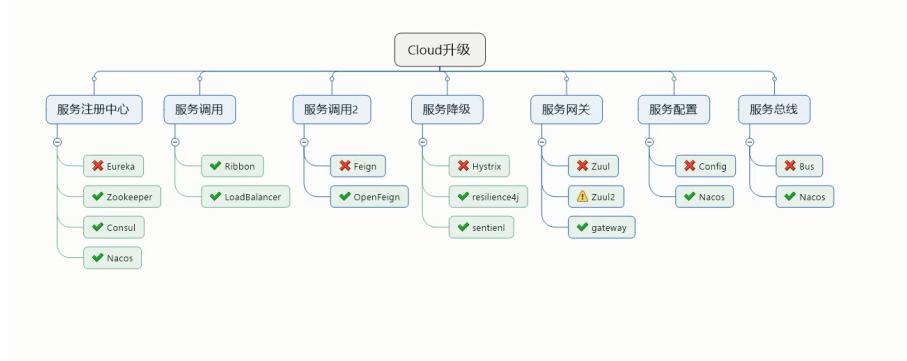
查看cloud版本和boot版本对应

<https://start.spring.io/actuator/info>

cloud组件停更说明

eureka停更 nacos强烈推荐使用

ribbon也停更了但还在使用



Eureka

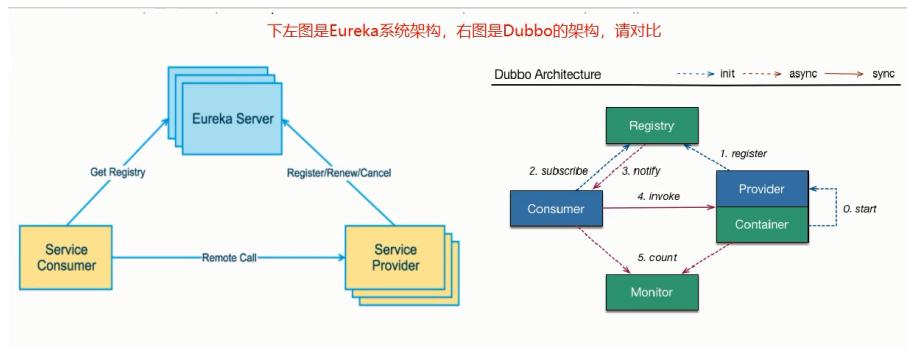
什么是服务治理

SpringCloud封装了netflix公司开发的Eurkea模块实现服务治理

在传统的rpc远程框架调用中，管理每个服务与服务之间的依赖关系比较复杂，管理比较复杂，所以需要服务治理，管理服务之间的依赖关系，可以实现服务调用、负载均衡、容错等，实现服务注册与发现。

Eureka采用了CS的设计架构，Eureka server作为服务注册功能的服务器，它是服务注册中心，而系统中的其他微服务，使用Eureka的客户端连接到Eureka Server并维持心跳连接。这样系统的维护人员就可以通过Eureka server来监控系统中各个微服务是否正常运行。

在服务注册与发现中，有一个注册中心，当服务器启动的时候，会把当前自己服务器的信息，比如服务地址 通讯地址等以别名的方式注册到注册中心上。另一方（消费者|服务提供者），以该别名的方式去注册中心获取到实际的服务通讯地址，然后再实现本地RPC调用远程RPC调用 核心设计思想：在于注册中心 因为使用注册中心管理每个服务于服务之间的一个依赖关系服务治理概念



Eureka包含两个组件：Eureka Server和Eureka Client

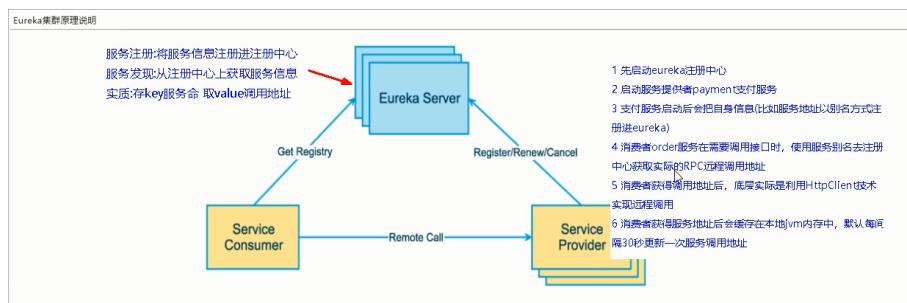
Eureka Server提供服务注册服务

各个微服务节点通过配置启动后，会在EurekaServer中进行注册，这样EurekaServer中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观看到。

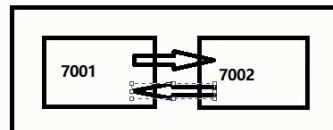
EurekaClient通过注册中心进行访问

是一个Java客户端，用于简化Eureka Server的交互，客户端同时也具备一个内置的、使用轮询(round-robin)负载算法的负载均衡器。在应用启动后，将会向Eureka Server发送心跳(默认周期为30秒)。如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，EurekaServer将会从服务注册表中把这个服务节点移除 (默认90秒)

Eureka集群原理

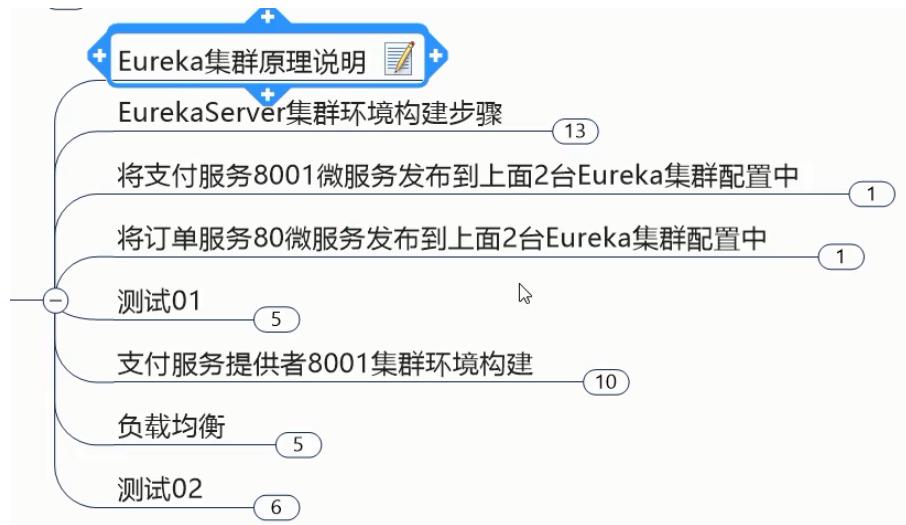


互相注册，相互守望



eureka

集群使用步骤：



Eureka服务发现

对于注册进Eureka里面的微服务，可以通过服务发现来获取该服务的信息

可以从官方提供的DiscoveryClient类中获取服务信息

```

@Autowired
private DiscoveryClient discoveryClient;

List<String>
clientServices=discoveryClient.getServices();

for (String clientService : clientServices) {
    List<ServiceInstance> serviceInstanceList =
discoveryClient.getInstances(clientService.toUpperCase());
    for (ServiceInstance serviceInstance :
serviceInstanceList) {

        log.info("serviceName
称:"+clientService+",host:"+serviceInstance.getHost()+",po
st:"+serviceInstance.getPort()+"uri:"+serviceInstance.get
Uri());
    }
}

```

需要在启动类中加上注解@EnableDiscoveryClient

Eureka自我保护机制

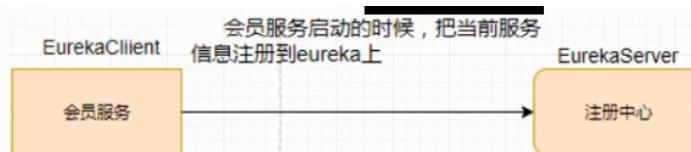
某时刻某一个微服务不可用了，Eureka不会立即清理，依旧会对该微服务信息进行保存

为什么会产生Eureka自我保护机制？

为了防止EurekaClient可以正常运行，但是与EurekaServer网络不通情况下，EurekaServer不会立刻将EurekaClient服务剔除

什么是自我保护模式？

默认情况下，如果EurekaServer在一定时间内没有接收到某个微服务实例的心跳，EurekaServer将会注销该实例（默认90秒）。但是当网络分区故障发生（延时、卡顿、拥挤）时，微服务与EurekaServer之间无法正常通信，以上行为可能变得非常危险了——因为微服务本身其实是健康的，此时本不应该注销这个微服务。Eureka通过“自我保护模式”来解决这个问题——当EurekaServer节点在短时间内丢失过多客户端时（可能发生了网络分区故障），那么这个节点就会进入自我保护模式。



自我保护机制：默认情况下EurekaClient定时向EurekaServer端发送心跳包

如果Eureka在server端在一定时间内（默认90秒）没有收到EurekaClient发送心跳包，便会直接从服务注册列表中剔除该服务，但是在短时间（90秒内）内丢失了大量的服务实例心跳，这时候EurekaServer会开启自我保护机制，不会剔除该服务（该现象可能出现在如果网络不通但是EurekaClient为出现宕机，此时如果换做别的注册中心如果一定时间内没有收到心跳会将剔除该服务，这样就出现了严重失误，因为客户端还能正常发送心跳，只是网络延迟问题，而保护机制是为了解决此问题而产生的）

在自我保护模式中，Eureka Server会保护服务注册表中的信息，不再注销任何服务实例。

它的设计哲学就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例。一句话讲解：好死不如赖活着

综上，自我保护模式是一种应对网络异常的安全保护措施。它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留）也不盲目注销任何健康的微服务。使用自我保护模式，可以让Eureka集群更加的健壮、稳定。

怎么禁止自我保护

在Eureka的服务端添加：

```
eureka:  
  server:  
    #关闭自我保护机制 保证不可用服务被及时剔除  
    enable-self-preservation: false  
    #接收客户端发送心跳的间隔时间 eviction 驱逐  
    eviction-interval-timer-in-ms: 2000
```

在Eureka的客户端添加：

```

eureka:
  instance:
    #心跳检测与续约时间 开发时可设置小些 方便及时剔除
    #eureka 客户端向服务器发送心跳的间隔时间 默认30秒
    lease-renewal-interval-in-seconds: 1
    #eureka 服务端在接收到最后一次心跳后等待时间上限，默认90秒 超时
    将剔除
    lease-expiration-duration-in-seconds: 2

```

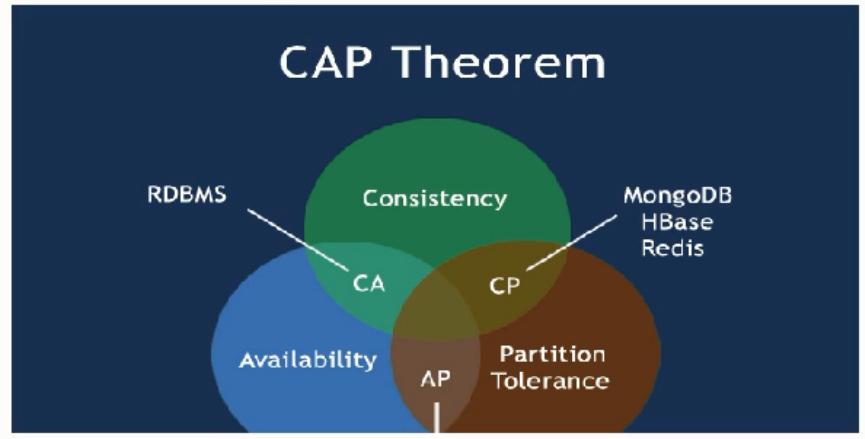
zookeeper注册中心

zookeeper是临时节点

consul

三个注册中心异同点:

组件名	语言	CAP	服务健康检查	对外暴露接口	Spring Cloud集成
Eureka	Java	AP	可配支持	HTTP	已集成
Consul	Go	CP	支持	HTTP/DNS	已集成
Zookeeper	Java	CP	支持	客户端	已集成



经典CAP图

最多只能同时较好的满足两个。

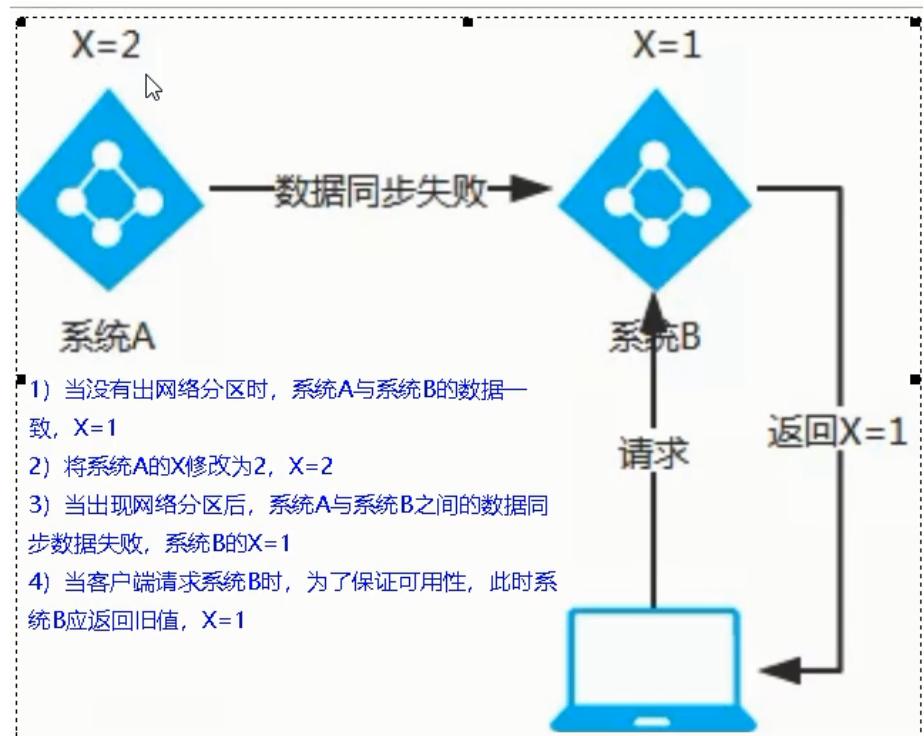
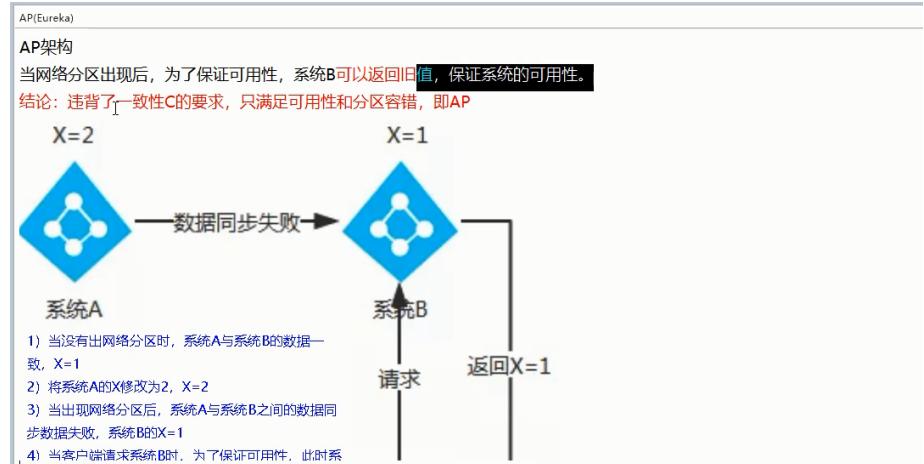
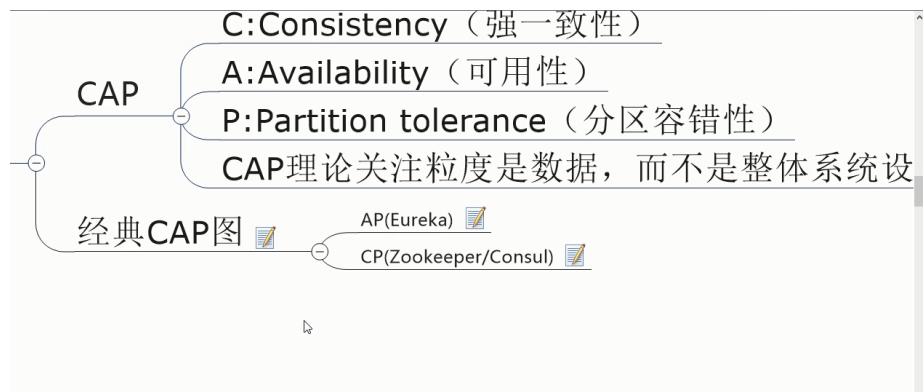
CAP理论的核心是：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求，

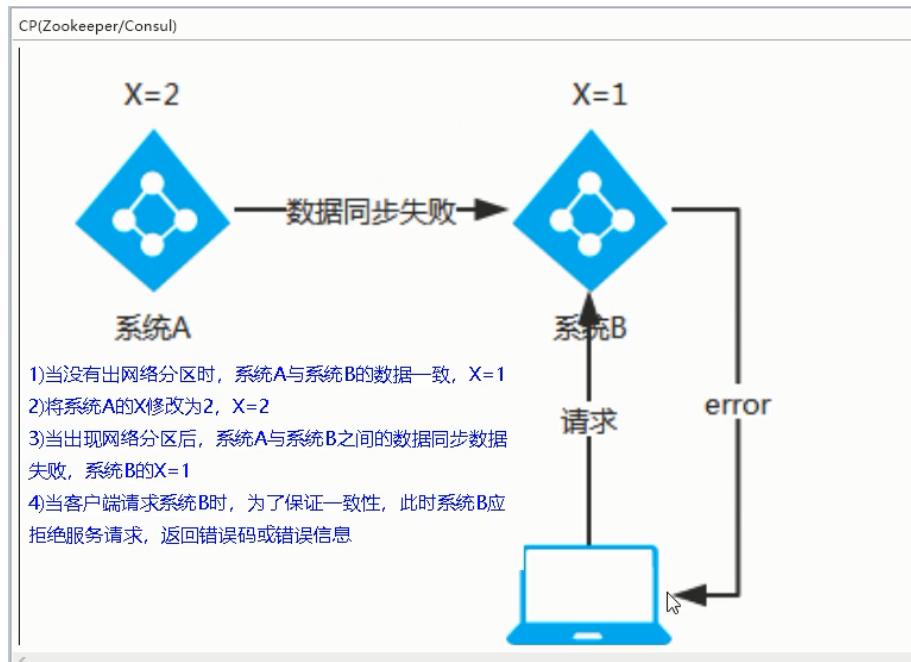
因此，根据 CAP 原理将 NoSQL 数据库分成了满足 CA 原则、满足 CP 原则和满足 AP 原则三 大类：

CA - 单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大。

CP - 满足一致性，分区容忍的系统，通常性能不是特别高。

AP - 满足可用性，分区容忍性的系统，通常可能对一致性要求低一些。





CP架构

当网络分区出现后, 为了保证一致性, 就必须拒接请求, 否则无法保证一致性

结论: 违背了可用性A的要求, 只满足一致性和分区容错, 即CP

Ribbon 负载均衡服务调用

是什么 :

Spring Cloud Ribbon是基于Netflix Ribbon实现的一套[客户端](#) **负载均衡的工具**。

简单的说, Ribbon是Netflix发布的开源项目, 主要功能是提供[客户端的软件负载均衡算法和服务调用](#)。Ribbon客户端组件提供一系列完善的配置项如连接超时, 重试等。简单的说, 就是在配置文件中列出Load Balancer (简称LB) 后面所有的机器, Ribbon会自动的帮助你基于某种规则(如简单轮询, 随机连接等)去连接这些机器。我们很容易使用Ribbon实现自定义的负载均衡算法。

Ribbon目前已经进入维护模式, 未来替换方案由springcloud提供的loadbandce

能做什么 :

LB (负载均衡)

LB负载均衡(Load Balance)是什么

简单的说就是将用户的请求平摊的分配到多个服务上，从而达到系统的HA（高可用）。

常见的负载均衡有软件Nginx，LVS，硬件F5等。

Ribbon本地负载均衡客户端 VS Nginx服务端负载均衡区别

Nginx是服务器负载均衡，客户端所有请求都会交给nginx，然后由nginx实现转发请求。即负载均衡是由服务端实现的。

Ribbon本地负载均衡，在调用微服务接口时候，会在注册中心上获取注册信息服务列表之后缓存到JVM本地，从而在本地实现RPC远程服务调用技术。

集中式LB：

即在服务的消费方和提供方之间使用独立的LB设施(可以是硬件，如F5，也可以是软件，如nginx)，由该设施负责把访问请求通过某种策略转发至服务的提供方；

进程式LB：

将LB逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择出一个合适的服务器。

Ribbon就属于进程内LB，它只是一个类库，集成于消费方进程，消费方通过它来获取到服务提供方的地址。

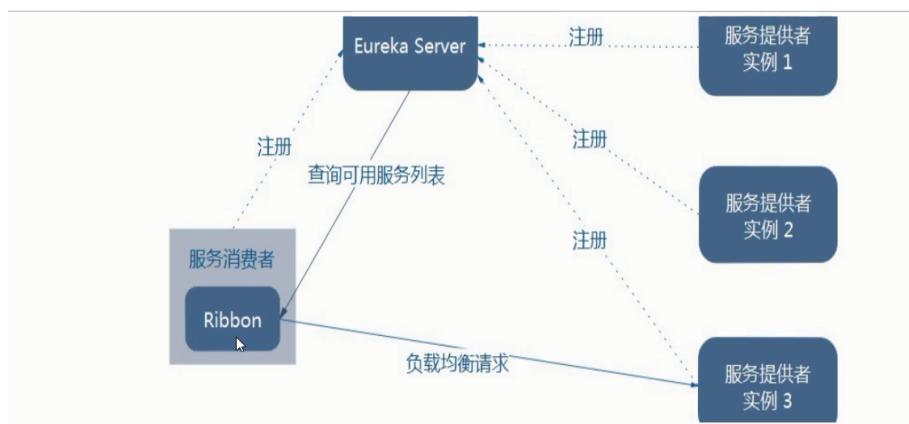
一句话概括：负载均衡+Resttemplate的调用

Ribbon负载均衡和Resttemplate的调用

Ribbon负载均衡演示：

架构说明：Ribbon其实就是一个软负载均衡的客户端组件，他可以和其他所需的请求客户端结合使用，如和Eureka结合是其中的一个实例。

Ribbon 架构：



Ribbon在工作时分成两步

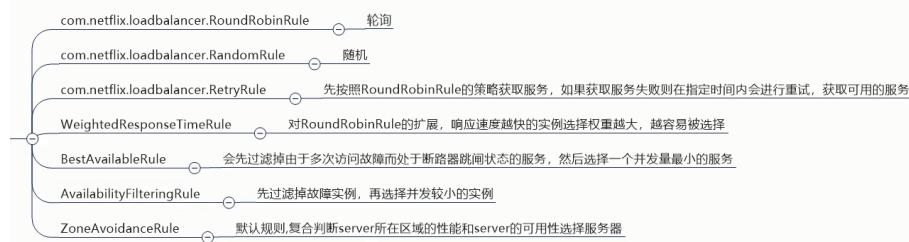
第一步先选择 EurekaServer , 它优先选择在同一个区域内负载较少的server.

第二步再根据用户指定的策略，在从server取到的服务注册列表中选择一个地址。

其中Ribbon提供了多种策略：比如轮询、随机和根据响应时间加权。

Ribbon的负载均衡实现:

IRule 根据特定算法从服务列表中选取一个要访问的服务，



注意配置细节

官方文档明确给出了警告：

这个自定义配置类不能放在@ComponentScan所扫描的当前包下以及子包下，

否则我们自定义的这个配置类就会被所有的Ribbon客户端所共享，达不到特殊化定制的目的了。[1]

Customizing the Ribbon Client

You can configure some bits of a Ribbon client using external properties in `<client>.ribbon.*`, which is no different than using the Netflix APIs natively, except that you can use Spring Boot configuration files. The native options can be inspected as static fields in `CommonClientConfigKey` (part of `ribbon-core`).

Spring Cloud also lets you take full control of the client by declaring additional configuration (on top of the `RibbonClientConfiguration`) using `@RibbonClient`. Example:

```
@Configuration
@RibbonClient(name = "foo", configuration = FooConfiguration.class)
public class TestConfiguration { }
```

默认负载均衡原理:

原理

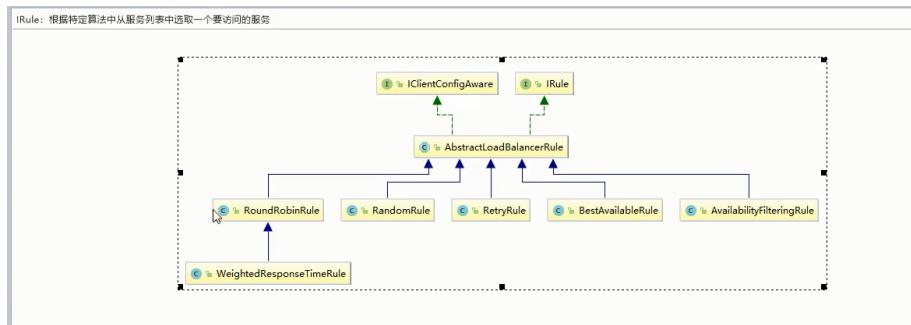
负载均衡算法：rest接口第几次请求数 % 服务器集群总数量 = 实际调用服务器位置下标，每次服务重启后rest接口计数从1开始。

```
List<ServiceInstance> instances = discoveryClient.getInstances("CLOUD-PAYMENT-SERVICE");
```

如：
List [0] instances = 127.0.0.1:8002
List [1] instances = 127.0.0.1:8001

8001+8002 组合成为集群，它们共计2台机器，集群总数为2，按照轮询算法原理：

当总请求数为1时：1 % 2 = 1 对应下标位置为1，则获得服务地址为127.0.0.1:8001
当总请求数为2时：2 % 2 = 0 对应下标位置为0，则获得服务地址为127.0.0.1:8002
当总请求数为3时：3 % 2 = 1 对应下标位置为1，则获得服务地址为127.0.0.1:8001
当总请求数为4时：4 % 2 = 0 对应下标位置为0，则获得服务地址为127.0.0.1:8002
如此类推.....



OpenFeign

openFeign是什么

<https://cloud.spring.io/spring-cloud-static/Hoxton.SR1/reference/htmlsingle/#spring-cloud-openfeign>

Feign是一个声明式WebService客户端。使用Feign能让编写Web Service客户端更加简单。它的使用方法是**定义一个服务接口然后在上面添加注解**。Feign也支持可拔插式的编码器和解码器。Spring Cloud对Feign进行了封装，使其支持了Spring MVC标准注解和HttpMessageConverters。Feign可以与Eureka和Ribbon组合使用以支持负载均衡。

6. Spring Cloud OpenFeign
Hoxton.SR1
This project provides OpenFeign integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.

6.1. Declarative REST Client: Feign
Feign is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud adds support for Spring MVC annotations and for using the same HttpMessageConverters used by default in Spring Web. Spring Cloud integrates Ribbon and Eureka, as well as Spring Cloud LoadBalancer to provide a load balanced http client when using Feign.

6.1.1. How to Include Feign

openFeign是一个声明式的web服务客户端，让编写web服务变得非常容易，只需创建一个接口，在接口上添加注解即可。

Feign能干什么
Feign旨在使编写Java Http客户端變得更容易。
前面在使用Ribbon+RestTemplate时，利用RestTemplate对http请求的封装处理，形成了一套模版化的调用方法。但是在实际开发中，由于对服务依赖的调用可能不止一处，往往一个接口会被多处调用，所以通常都会针对每个微服务自行封装一些客户端类来包装这些依赖服务的调用。所以，Feign在此基础上做了进一步封装，由他来帮助我们定义和实现依赖服务接口的定义。在Feign的实现下，我们只需创建一个接口并使用注解的方式来配置它（以前是Dao接口上面标注Mapper注解，现在是一个微服务接口上面标注一个Feign注解即可），即可完成对服务提供方的接口绑定，简化了使用Spring cloud Ribbon时，自动封装服务调用客户端的开发量。

Feign集成了Ribbon
利用Ribbon维护了Payment的服务列表信息，并且通过轮询实现了客户端的负载均衡。而与Ribbon不同的是，**通过feign只需要定义服务绑定接口且以声明式的方法**，优雅而简单的实现了服务调用。

Feign和open Feign的区别：

Feign和OpenFeign两者区别

Feign	OpenFeign
Feign是Spring Cloud组件中的一个轻量级RESTful的HTTP服务客户端 Feign内置了Ribbon，用来做客户端负载均衡，去调用服务注册中心的服务。Feign的使用方式是：使用Feign的注解定义接口，调用这个接口，就可以调用服务注册中心的服务	OpenFeign是Spring Cloud 在Feign的基础上支持了SpringMVC的注解，如@RequestMapping等等。OpenFeign的@FeignClient可以解析SpringMVC的@RequestMapping注解下的接口，并通过动态代理的方式产生实现类，实现类中做负载均衡并调用其他服务。
<dependency><groupId>org.springframework.cloud</groupId><artifactId>spring-cloud-starter-feign</artifactId></dependency>	<dependency><groupId>org.springframework.cloud</groupId><artifactId>spring-cloud-starter-openfeign</artifactId></dependency>

openFeign使用步骤：

微服务调用接口+@FeignClient

主启动类上添加 @EnableFeignClient

业务类:

业务逻辑接口+@FeignClient配置provider服务

新建PaymentFeignService接口并新增注解@FeignClient

OpenFeign超时控制

默认openFeign客户端超时等待只有1秒钟

openFeign日志增强:

日志级别

NONE: 默认的，不显示任何日志；

BASIC: 仅记录请求方法、URL、响应状态码及执行时间；

HEADERS: 除了 BASIC 中定义的信息之外，还有请求和响应的头信息；

FULL: 除了 HEADERS 中定义的信息之外，还有请求和响应的正文及元数据。

配置日志bean

```
package com.atguigu.springcloud.cfgbeans;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import feign.Logger;

/**
 * @author zzyy
 * @create 2019-11-10 17:00
 */
@Configuration
public class FeignConfig
{
    @Bean
    Logger.Level feignLoggerLevel()
    {
        return Logger.Level.FULL;
    }
}
```

#feign 日志以什么级别监控那个接口

```
logging:
  level:
    com.george.springcloud.service: debug
```

Hystrix

概述

分布式系统面临的问题

复杂分布式体系结构中的应用程序有数十个依赖关系，每个依赖关系在某些时候将不可避免失败。

分布式系统面临的问题

服务雪崩

多个微服务之间调用的时候，假设微服务A调用微服务B和微服务C，微服务B和微服务C又调用其它的微服务，这就是所谓的“**扇出**”。如果扇出的链路上某个微服务的调用响应时间过长或者不可用，对微服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃，所谓的“**雪崩效应**”。

对于高流量的应用来说，单一的后端依赖可能会导致所有服务器上的所有资源都在几秒钟内饱和。比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，备份队列，线程和其他系统资源紧张，导致整个系统发生更多的级联故障。这些都表示需要对故障和延迟进行隔离和管理，以便单个依赖关系的失败，不能取消整个应用程序或系统。

所以，

通常当你发现一个模块下的某个实例失败后，这时候这个模块依然还会接收流量，然后这个有问题的模块还调用了其他的模块，这样就会发生级联故障，或者叫雪崩。

Hystrix定义

Hystrix是一个用于处理分布式系统的**延时**和**容错**的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，Hystrix能够保证在一个依赖出问题的情况下，**不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性**。

“断路器”本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），**向调用方返回一个符合预期的、可处理的备选响应（FallBack）**，而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要地占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。

Hystrix作用

服务降级、服务熔断、接近实时监控

Hystrix重要概念

服务降级 fallback

服务繁忙，请稍后再试 不要让客户端等待并立刻返回一个友好提示

哪些情况会出现？

程序运行异常、超时、服务熔断

服务熔断 break

类比保险丝达到最大服务访问后，直接拒绝访问，拉闸限电，然后调用服务降级的方法并返回友好提示

服务限流 flowlimit

秒杀高并发等操作，严禁一窝蜂的过来拥挤，大家排队，一秒钟N个，有序进行

Hystrix案例

构建

step1: 新建maven项目 cloud-provider-hystrix-payment8001

pom.xml配置如下：

```
<dependencies>

    <!-- hystrix -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-
hystrix</artifactId>
    </dependency>

    <!-- 引入eureka客户端 -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-
eureka-client</artifactId>
    </dependency>

    <dependency>
        <groupId>com.telecom.springcloud</groupId>
        <artifactId>cloud-api-commons</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-
web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-
actuator</artifactId>
    </dependency>

    <!--
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
```

```
        <artifactId>mybatis-spring-boot-
starter</artifactId>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid-spring-boot-
starter</artifactId>
            <version>1.1.10</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-
jdbc</artifactId>
            </dependency> -->

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <optional>true</optional>
        </dependency>

    </dependencies>
```

step2:修改application.yml配置

```
server:
  port: 8001
spring:
  application:
    name: cloud-provider-hystrix-payment
#eureka配置
eureka:
  client:
    #表示将自己注册到Eureak Server 默认为true
    register-with-eureka: true
    #是否从EureakServer中抓取已有的注册信息 默认为true 单节点无所谓 集群必须设置为true , 才能配合ribbon使用负载均衡
    fetch-registry: true
```

```
service-url:  
    #单机版 Eureka  
    defaultZone: http://eureka7001:7001/eureka
```

step3: 主启动类、 controller层、 service层

主启动类:

```
package com.telecom.springcloud;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
  
/**  
 * Date : 2020-11-09 10:32  
 * Description :  
 */  
@SpringBootApplication  
@EnableEurekaClient  
public class PaymentHystrixMain8001 {  
    public static void main(String[] args) {  
  
        SpringApplication.run(PaymentHystrixMain8001.class,args);  
    }  
}
```

service层:

```
package com.telecom.springcloud.service;  
  
import org.springframework.stereotype.Service;  
  
import java.util.concurrent.TimeUnit;  
  
/**  
 * Date : 2020-11-09 10:39  
 * Description :  
 */  
@Service  
public class PaymentService {
```

```

    /**
     * 服务正常
     * @param id
     * @return
     */
    public String paymentInfo_ok(Integer id){
        return "线程池： " +
Thread.currentThread().getName() + " PaymenyInfo_OK,id: "
+ id + "\t" + "O(∩_∩)O哈哈~";
    }

    /**
     * 服务超时
     * @param id
     * @return
     */
    public String paymentInfo_timeout(Integer id){
        int timeNumber = 3000;
        try {
            TimeUnit.MILLISECONDS.sleep(timeNumber);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "线程池： " +
Thread.currentThread().getName() +
PaymenyInfo_TimeOut,id: " + id + "\t" + "O(∩_∩)O哈哈~" +
耗时" + timeNumber + "毫秒";
    }
}

```

controller层：

```

package com.telecom.springcloud.controller;

import com.telecom.springcloud.service.PaymentService;
import lombok.extern.slf4j.Slf4j;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.PathVariable;
import
org.springframework.web.bind.annotation.RestController;

/**
 * Date : 2020-11-09 10:39

```

```
* Description:  
*/  
@RestController  
@Slf4j  
public class PaymentController {  
  
    @Autowired  
    private PaymentService paymentService;  
  
    @GetMapping("/payment/hystrix/ok/{id}")  
    public String paymentInfo_ok(@PathVariable("id")  
Integer id){  
        String result = paymentService.paymentInfo_ok(id);  
        log.info("=====result===="+result);  
        return result;  
    }  
  
    @GetMapping("/payment/hystrix/timeout/{id}")  
    public String paymentInfo_timeout(@PathVariable("id")  
Integer id){  
        String result =  
paymentService.paymentInfo_timeout(id);  
        log.info("=====result===="+result);  
        return result;  
    }  
}
```

step4:测试

启动eureka7001

启动cloud-provider-hystrix-payment8001

访问：<http://localhost:8001/payment/hystrix/ok/1> (服务正常)

<http://localhost:8001/payment/hystrix/timeout/1> (服务超时)

高并发测试

测试步骤:使用jmeter模拟并发请求

用jmeter模拟20000个请求去访问<http://localhost:8001/payment/hystrix/timeout/1>，此时在浏览器访问<http://localhost:8001/payment/hystrix/ok/1> 和<http://localhost:8001/payment/hystrix/timeout/1>

结果：<http://localhost:8001/payment/hystrix/ok/1>访问会变慢，当然<http://localhost:8001/payment/hystrix/timeout/1>会更慢

变慢原因：tomcat的默认工作线程数被打满了，没有多余的线程来分解压力和处理

订单微服务调用支付微服务出现卡顿

新建订单微服务模块

pom.xml

```
<dependencies>

    <!-- open feign      -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-openfeign</artifactId>
    </dependency>

    <!-- hystrix      -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
    </dependency>

    <!-- 引入自定义的api通用包，可以使用Payment支付Entity      -->
    <dependency>
        <groupId>com.telecom.springcloud</groupId>
        <artifactId>cloud-api-commons</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
    <!-- 引入eureka客户端      -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-  
actuator</artifactId>  
</dependency>  
  
<dependency>  
    <groupId>org.projectlombok</groupId>  
    <artifactId>lombok</artifactId>  
</dependency>  
  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-devtools</artifactId>  
    <optional>true</optional>  
</dependency>  
  
</dependencies>
```

yml配置文件:

```
server:  
  port: 80  
spring:  
  application:  
    name: cloud-consumer-feign-hystrix-order  
eureka:  
  client:  
    #是否将自己注册到注册中心， 默认true  
    register-with-eureka: true  
    #是否从EurekaServer抓取已有的注册信息，单机无所谓，集群必须设置  
    #为true配合ribbon使用负载均衡  
    fetch-registry: true  
    service-url:  
      defaultZone: http://eureka7001:7001/eureka #单机版
```

主启动类:

```
import org.springframework.boot.SpringApplication;  
import  
org.springframework.boot.autoconfigure.SpringBootApplication;  
import  
org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
import  
org.springframework.cloud.openfeign.EnableFeignClients;  
  
/**
```

```
* Date : 2020-11-09 15:51
* Description :
*/
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
public class FeignHystrixorder80 {

    public static void main(String[] args) {

        SpringApplication.run(FeignHystrixOrder80.class,args);
    }

}
```

业务层:

```
package com.telecom.springcloud.service;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.stereotype.Service;
import org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.PathVariable;

/**
 * Date : 2020-11-09 15:53
 * Description :
 */
@FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT")
@Service
public interface PaymentHystrixService {

    @GetMapping("/payment/hystrix/ok/{id}")
    public String paymentInfo_ok(@PathVariable("id")
Integer id);

    @GetMapping("/payment/hystrix/timeout/{id}")
    public String paymentInfo_timeout(@PathVariable("id")
Integer id);
}
```

controller层:

```
package com.telecom.springcloud.controller;
```

```

import
com.telecom.springcloud.service.PaymentHystrixService;
import lombok.extern.slf4j.Slf4j;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.PathVariable;
import
org.springframework.web.bind.annotation.RestController;

/**
 * Date : 2020-11-09 15:58
 * Description :
 */
@RestController
@Slf4j
public class OrderHystrixController {

    @Autowired
    private PaymentHystrixService paymentHystrixService;

    @GetMapping("/consumer/payment/hystrix/ok/{id}")
    public String paymentInfo_ok(@PathVariable("id")
Integer id){
        String result =
paymentHystrixService.paymentInfo_ok(id);
        log.info("order-hystrix-80 paymentInfo_ok
result:"+result);
        return result;
    }

    @GetMapping("/consumer/payment/hystrix/timeout/{id}")
    public String paymentInfo_timeout(@PathVariable("id")
Integer id){
        String result =
paymentHystrixService.paymentInfo_timeout(id);
        log.info("order-hystrix-80 paymentInfo_timeout
result:"+result);
        return result;
    }
}

```

20000个线程请求payment8001,消费者80侧再去访问<http://localhost/consumer/payment/hystrix/ok/1> 消费者侧会出现等待(转圈圈)或者直接报超时异常。

故障现象和导致原因:

8001的同一层次的其他接口微服务直接被困死，因为tomcat线程池里面的工作线程已经被挤占完毕。80此时调用8001，客户端响应缓慢。

如何解决？

超时导致服务器变慢 -----超时不再等待

出错（程序宕机或出错）-----要兜底

解决思路：

对方服务（8001）超时了，调用者（80）不能一直卡死等待，必须有服务降级

对方服务（8001）down机了，调用者（80）不能一直卡死等待，必须有服务降级

对方服务（8001）OK，调用者（80）自己出故障或有自我要求（自己的等待时间小于服务提供者），自己处理降级

Hystrix服务降级之支付侧fallback

通过@HystrixCommand降低配置

设置自身调用超时时间的峰值，峰值内可以正常运行，超过了需要有兜底的方法处理，作服务降级fallback

实现：

业务侧 在方法上添加@HystrixCommand

```
 /**
 * 服务超时
 * @param id
 * @return
 */
@HystrixCommand(fallbackMethod =
"paymentInfo_timeoutHandler", commandProperties = {
    @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value =
"3000")//三秒钟以内就是正常逻辑
})
public String paymentInfo_timeout(Integer id){
    int timeNumber = 3000;
    try {
        TimeUnit.MILLISECONDS.sleep(timeNumber);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

```

        }
        return "线程池： " +
Thread.currentThread().getName() +
PaymenyInfo_TimeOut,id: " + id + "\t" + "o(∩_∩)o哈哈~" + "
耗时" + timeNumber + "毫秒";
    }

    public String paymentInfo_timeoutHandler(Integer id){
        return "线程
池：" + Thread.currentThread().getName() + " 系统繁忙， 请稍候再
试 ,id: " + id + "\t" + "哭了哇呜";
    }
}

```

主启动类上添加 @EnableCircuitBreaker

Hystrix服务降级之订单侧fallback

通常，服务降级都是在客户端侧进行的

controller添加@HystrixCommand

```

@GetMapping("/consumer/payment/hystrix/timeout/{id}")
@HystrixCommand(fallbackMethod =
"paymentTimeOutFallbackMethod", commandProperties = {
    @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value =
"3000")
})
public String paymentInfo_timeout(@PathVariable("id")
Integer id){
    String result =
paymentHystrixService.paymentInfo_timeout(id);
    log.info("order-hystrix-80 paymentInfo_timeout
result:" + result);
    return result;
}

//兜底方法
public String
paymentTimeOutFallbackMethod(@PathVariable("id") Integer
id){
    return "我是消费者80，对付支付系统繁忙请10秒钟后再试或者自
己运行出错请检查自己，(╥_╥)";
}

```

主启动类添加 @EnableHystrix

Hystrix全局服务降级

controller层要降级处理的类上添加注解@DefaultProperties

```
@DefaultProperties(defaultFallback =  
    "paymentGlobalFallbackMethod")
```

添加方法:

```
public String paymentGlobalFallbackMethod(){  
    return "全局报错异常, (T_T)";  
}
```

controller层中的方法上仍然要加注解@HystrixCommand

Hystrix通配服务降级

创建实现类PaymentFallbackService去实现抽象接口PaymentHystrixService

PaymentHystrixService:

```
@FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT")  
@Service  
public interface PaymentHystrixService {  
  
    @GetMapping("/payment/hystrix/ok/{id}")  
    public String paymentInfo_ok(@PathVariable("id")  
        Integer id);  
  
    @GetMapping("/payment/hystrix/timeout/{id}")  
    public String paymentInfo_timeout(@PathVariable("id")  
        Integer id);  
}
```

PaymentFallbackService:

```
@Component
public class PaymentFallbackService implements
PaymentHystrixService {
    @Override
    public String paymentInfo_ok(Integer id) {
        return "-----PaymentFallbackService fall back-
[paymentInfo_OK],o(╥﹏╥)o";
    }

    @Override
    public String paymentInfo_timeout(Integer id) {
        return "-----PaymentFallbackService fall back-
[paymentInfo_TimeOut],o(╥﹏╥)o";
    }
}
```

在抽象类PaymentHystrixService上的@FeignClient注解中定义fallback指定的类

```
@FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-
PAYMENT", fallback = PaymentFallbackService.class)
```

注意在订单侧的controller层的方法中不需要添加注解@HystrixCommand

Hystrix服务熔断

服务熔断机制概述

熔断机制是应对雪崩效应的一种微服务链路保护机制。当扇出链路的某个微服务出错不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用。快速返回错误的响应信息。**当检测到该节点微服务调用正常后，恢复调用链路。**

在SpringCloud中，熔断机制通过Hystrix实现。Hystrix会监控微服务调用的状况，当失败的调用到一定的阈值，缺省是5秒内调用20次失败，就会启动熔断机制。熔断机制的注解是@HystrixCommand

```
//服务熔断
@HystrixCommand(fallbackMethod =
"paymentCircuitBreaker_fallback", commandProperties = {
    @HystrixProperty(name =
"circuitBreaker.enabled", value = "true"), //是否开启断路器
    @HystrixProperty(name =
"circuitBreaker.requestVolumeThreshold", value = "10"), //请求次数
    @HystrixProperty(name =
"circuitBreaker.sleepWindowInMilliseconds", value =
```

```

    "10000"), //时间范围
    @HystrixProperty(name =
        "circuitBreaker.errorThresholdPercentage", value = "60"), //
        失败率达到多少后跳闸
    })
    public String paymentCircuitBreaker(@PathVariable("id")
Integer id){
    if (id < 0){
        throw new RuntimeException("*****id 不能负数");
    }
    String serialNumber = IdUtil.simpleUUID();

    return Thread.currentThread().getName()+"\t"+调用成
功,流水号 : "+serialNumber;
}
public String
paymentCircuitBreaker_fallback(@PathVariable("id") Integer
id){
    return "id 不能负数,请稍候再试,(ㄒ_ㄒ)/~~      id: "
+id;
}

```

服务熔断原理总结:

熔断类型：

熔断打开：请求不再进行调用当前服务，内部设置时钟一般为MTTR(平均故障处理时间)，当打开时长达到所设时钟则进入熔断状态

熔断关闭：熔断关闭不会对服务进行熔断

熔断半开：部分请求根据规则调用当前服务，如果请求成功且符合规则则认为当前服务恢复正常，关闭熔断

断路器在什么情况下启用：

```

//=====服务熔断
@HystrixCommand(fallbackMethod = "paymentCircuitBreaker_fallback", commandProperties = {
    @HystrixProperty(name = "circuitBreaker.enabled", value = "true"), //开启断路器
    @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10"), //请求次数超过了峰值, 熔断器将会从关闭
    @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "10000"), //时间范围
    @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60"), //失败率达到多少后跳闸
})
public String paymentCircuitBreaker(@PathVariable("id") Integer id)

```

涉及到断路器的三个重要参数：快照时间窗、请求总数阀值、错误百分比阀值。

1: 快照时间窗：断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间窗，默认为最近的10秒。

2: 请求总数阀值：在快照时间窗内，必须满足请求总数阀值才有资格熔断。默认为20，意味着在10秒内，如果该hystrix命令的调用次数不足20次，即使所有的请求都超时或其他原因失败，断路器都不会打开。

3: 错误百分比阀值：当请求总数在快照时间窗内超过了阀值，比如发生了30次调用，如果在这30次调用中，有15次发生了超时异常，也就是超过50%的错误百分比，在默认设定50%阀值情况下，这时候就会将断路器打开。

断路器开启或关闭的条件：

1. 当满足一定阀值的时候（默认10秒内超过20个请求次数）

2. 当失败率达到一定的时候（默认10秒内超过50%请求失败）
3. 到达以上阀值，断路器将会开启
4. 当开启的时候，所有请求都不会进行转发
5. 一段时间之后（默认是5秒），这个时候断路器是半开状态，会让其中一个请求进行转发。如果成功，断路器会关闭，若失败，继续开启。重复4和5

断路器开启之后：

1: 再有请求调用的时候，将不会调用主逻辑，而是直接调用降级fallback。通过断路器，实现了自动地发现错误并将降级逻辑切换为主逻辑，减少响应延迟的效果。

2: 原来的主逻辑要如何恢复呢？

对于这一问题，hystrix也为我们实现了自动恢复功能。

当断路器打开，对主逻辑进行熔断之后，hystrix会启动一个休眠时间窗，在这个时间窗内，降级逻辑是临时的成为主逻辑，

当休眠时间窗到期，断路器将进入半开状态，释放一次请求到原来的逻辑上，如果此次请求正常返回，那么断路器将继续闭合，

主逻辑恢复，如果这次请求依然有问题，断路器继续进入打开状态，休眠时间窗重新计时。

断路器的所有配置：

```

@HystrixProperty(name = "metrics.rollingPercentile.bucketSize", value = "100"),
    // 该属性用来设置采集影响断路器状态的健康快照（请求的成功、 错误百分比）的间隔等待时间。
@HystrixProperty(name = "metrics.healthSnapshot.intervalInMilliseconds", value = "500"),
    // 是否开启请求缓存
@HystrixProperty(name = "requestCache.enabled", value = "true"),
    // HystrixCommand 的执行和事件是否打印日志到 HystrixRequestLog 中
@HystrixProperty(name = "requestLog.enabled", value = "true"),
},
threadPoolProperties = {
    // 该参数用来设置执行命令线程池的核心线程数，该值也就是命令执行的最大并发量
@HystrixProperty(name = "coreSize", value = "10"),
    // 该参数用来设置线程池的最大队列大小。当设置为 -1 时，线程池将使用 SynchronousQueue 实现的队列,
    // 否则将使用 LinkedBlockingQueue 实现的队列。
@HystrixProperty(name = "maxQueueSize", value = "-1"),
    // 该参数用来为队列设置拒绝阈值。通过该参数，即使队列没有达到最大值也能拒绝请求。
    // 该参数主要是对 LinkedBlockingQueue 队列的补充，因为 LinkedBlockingQueue
    // 队列不能动态修改它的对象大小，而通过该属性就可以调整拒绝请求的队列大小了。
@HystrixProperty(name = "queueSizeRejectionThreshold", value = "5"),
}
)
public String strConsumer() {
    return "hello 2020";
}
}

```

Hystrix服务熔断工作流程

1	根据 HystrixCommand (用在依赖的服务返回单个操作结果的时候) 或 HystrixObservableCommand (用在依赖的服务返回多个操作结果的时候) 对象。
2	命令执行。其中 HystrixCommand 实现了下图前两种执行方式，而 HystrixObservableCommand 实现了后两种执行方式：executed()：同步执行，从依赖的服务返回一个单一的结果对象，或是在发生错误的时候抛出异常；queue()：异步执行，直接返回一个 Future<Object>，其中包含了阶段执行结果时返回的单一结果对象，observer()：返回 Observable<Object>，它代表了操作的多个结果，它是一个 Hot Observable (不设有“订阅者”，它会在被调用后对事件进行发布，所以对于 Hot Observable 的每一个“订阅者”都可能是从“事件流”的中途开始的，并可能只是看到了整个操作的尾部数据)，toObservable()：同样会返回 Observable<Object>，也代表了操作的多个结果，但它返回的是一个Cold Observable (设有“订阅者”的时候并不会发布事件，而是通过等待，直到有“订阅者”之后才发布事件，所以对于 Cold Observable 的订阅者，它可以保证从一开始看到整个操作的全部数据)。
3	若当前命令的请求缓存功能是被启用的，并且该命令处于缓存中，那么缓存的结果会立即以 Observable<Object> 的形式返回。
4	检查断路器是否是打开状态，如果断路器是打开的，那么Hystrix不会执行命令，而是转接到 fallback 处理逻辑（第 8 步）；如果断路器是关闭的，检查是否有可用资源来执行命令（第 5 步）。
5	线程池/请求队列/信号量是否占满。如果命令依赖服务的专属线程池和请求队列，或者信号量（不使用线程池的时候）已经被占满，那么 Hystrix 也不会执行命令，而是转接到 fallback 处理逻辑（第 8 步）。
6	Hystrix 会根据我们编写的规则来决定采取什么样的方式请求依赖服务，HystrixCommand.run()：返回一个单一的结果，或者抛出异常，HystrixObservableCommand.construct()：返回一个 Observable<Object> 对象来发射多个结果，或者通过 onError 发送错误通知。
7	Hystrix 会将“成功”、“失败”、“拒绝”、“超时”等信息报告给断路器，而断路器会维护一组计数器来统计这些数据。断路器会使用这些统计数据来决定是否要将断路器打开，来对某个依赖服务的请求进行“熔断/回落”。
8	当命令执行失败的时候，Hystrix 会进入 fallback 尝试回退逻辑，我们通常也称这操作为“聚类降级”，而能够引起服务降级处理的情况有下面几种：第4步：当前命令处于“熔断/短路”状态，断路器是打开的时候，第5步：当前命令的本地“请求队列/信号量被占满”的时候，第6步：HystrixObservableCommand.construct() 或 HystrixCommand.run() 抛出异常的时候。
9	当Hystrix命令执行成功之后，它会将处理结果返回或是以Observable 的形式返回。

tips：如果我们没有为命令实现降级逻辑或者在继承处理器中抛出了异常，Hystrix 依然会返回一个 Observable 对象，但是它不会发射任何结果数据，而是通过 onError 方法通知命令立即中断请求，并通过 onError 方法将引起命令失败的异常发送给调用者。

Hystrix服务监控hystrixDashBoard

修改修改cloud-provider-hystrix-payment8001 ,新版本Hystrix需要在主启动类 MainAppHystrix8001中指定监控路径

```
@Bean
public ServletRegistrationBean getServlet(){
    HystrixMetricsStreamServlet streamServlet = new
    HystrixMetricsStreamServlet();
    ServletRegistrationBean registrationBean = new
    ServletRegistrationBean(streamServlet);
    registrationBean.setLoadOnStartup(1);
    registrationBean.addUrlMappings("/hystrix.stream");

    registrationBean.setName("HystrixMetricsStreamServlet");
    return registrationBean;
}
```

监控测试：

启动1个Eureka或3个Eureka即可

填写监控地址 9001监控8001 <http://localhost:8001/hystrix.stream>

